SOFTWARE PROGRAMMER'S MANUAL (SPM)

FOR THE

“Virtual Target API”

OF CSCI

“OBA Virtual Target Supplies”

for Windows NT

CONTRACT No E011007

CDRL SEQUENCE No 7b

Prepared for:
THALES Université Coopération, Buc

Prepared by:
Byzance Informatique, Grigny

Authenticated by		Approved by	
	Thalès Université Coopération		Byzance Informatique
Date		Date	

	NUMERO DOCUMENT / DOCUMENT	FORMAT /	PAGE
THALES UNIVERSITE COOPERATION	**56 699 445-508**	**A4**	**1/16**
XXXX		A -	REV

CHANGES

REVISION	DESCRIPTION
A	p1 : Thalès Logo, CDRL N° All pages : CSCI N°
B	
C	
D	
E	
F	

ind. + Date	– 17/01/2002	A 28/02/2002	B	C	D	E	F
Written by	D. Boucon	D. Boucon					
Checked by	D. Jeanjean						
Approved by	D. Boucon						

TABLE OF CONTENTS

1. SCOPE

1.1 IDENTIFICATION

CSCI “OBA Virtual Target Supplies” (OBA_VTS) Identification Number: 56 699 445.

The Virtual Target API is one of the parts of this CSCI.

1.2 CSCI OVERVIEW

The CSCI “OBA Virtual Target Supplies” (OBA_VTS) is intended to give a simulation environment for a student’s development project, named “OBA software”.

This complete environment (OBA_VTS) for executing and testing OBA software is a kind of virtual target built upon the operating system of the workstation used for OBA software development. It allows for running the OBA software on a development host machine, instead of the real target.

The Virtual Target API is the description of the programming interface between the OBA software and the OBA Testbench.

1.3 SYSTEM OVERVIEW

The system is the complete set of necessary materials for the pedagogical software development project made by teams of students, including this CSCI.

1.4 DOCUMENT OVERVIEW

This document is intended for OBA programmers (i.e. students). It explains them how to make use of OBA target resources within their programs. It defines an application programming interface level of theses resources.

This document gives details about the data and function calls of the API and explains their correct and necessary usages.

2. REFERENCED DOCUMENTS

[IRS] Interfaces Requirements Specification of Safe Drive Subsystem, 31 000 100-506

[SUM] OBA Testbench Software Virtual Target API Programming Manual, 56 699 445-108

3. SOFTWARE PROGRAMMING ENVIRONMENT

The development host platform (hardware and software) used by students is :

- a PC with Windows NT,
- the C++ programming language, with Visual C++ (from Microsoft) as the development tool,
- the specific testbench (called “OBA Testbench”) dedicated to the project (for the use of this testbench, see *OBA Testbench Software User’s Manual* [SUM]),
- the OBA Virtual Target libraries and include files
- a UML tool (such as Rose, Rhapsody, Objecteering or StP),
- an office automation tool (such as Office, StarOffice,…)

4. PROGRAMMING INFORMATION

4.1 DESCRIPTION OF CLASS : OBATARGETMACHINE

This class is a software simulation of the **OBATarget** hardware of the **SafeDrive** project. It should be considered as some kind of a "virtual" micro-controller board, coming along with its BSP (Board Support Package) that gives access to the target resources.

Note : There is no need (nor possibility) to create instances of this class, as it represents something unique and external to the application software. C

- ```
  private:
        OBATargetMachine() {;}
  ```

Class constructor is made private to deter instantiation compulsions, since OBATargetMachine is an all-static class, so, conceptually, a single object by itself.

To get access to this single object, just use :

```
#include "OBATargetMachine.h"
```

and then, call static member functions using class scope operator, e.g.:

```
OTM::setPeriodicTimer (200, 10);
```

4.2 PORTS.

Communications with sensors and actuators in the physical environment are done through ports by reading from and writing to related buffers.

The ports are grouped by interfaces (refer to *Safe Drive Interfaces Requirements Specification* [IRS] for detailed layout) :

Cruising automobile to OBA

- `static void * pulseCounterWordAddress;`

  address of the buffer containing the pulse counter word

- `static void * throttleResponseWordAddress;`

  address of the buffer containing the throttle response word.

Driver to OBA

- `static void * drivingStationInterruptWordAddress;`

  address of the buffer containing the driving station interrupt word.

- `static void * drivingStationStateWordAddress;`
```

address of the buffer containing the driving station state word.

- `static void * controlPanelInterruptWordAddress;`

  address of the buffer containing the control panel interrupt word.

OBA to Cruising automobile

- `static void * throttleCommandWordAddress;`

  address of the buffer containing the throttle command word.

OBA to Driver:

- `static void * displayBufferAddress;`

  address of the buffer containing the display message.

- `static void * ledCommandByteAddress;`

  address of the buffer containing the led command byte.

4.3 INTERRUPTS

The target micro-controller board is able to manage interrupts : an interrupt is nothing more than a register offset in a register set, called the interrupt vector, that has special execution semantics.

- `typedef unsigned Interrupt;`

- ```
  #define lastSlotOffset    15
  #define interruptCardinal lastSlotOffset+1
  static InterruptVectorSlot interruptVector [interruptCardinal];
  ```

Each register in the interrupt vector is called an interrupt vector slot : the micro-controller is able to manage as many interrupts as there are slots in the interrupt vector.

An interrupt vector slot is a simulation of a very special hardware register and so, have special semantics :

- Instance cannot be created !! (it is not a software entity, but the software representation of a hardware entity),
- an `InterruptHandler` (see below) can be loaded in it, or transfer the contents of another interrupt vector slot, or also reset it (i.e. unset its interrupt handler).

Warning : ***After resetting an interrupt vector slot, a hit of the corresponding interrupt will stall the program, unless the corresponding bit in the enable mask register had been cleared.***

- ```
  class InterruptVectorSlot {
  …
  public  :
        void operator= (InterruptHandler);
        void operator= (const InterruptVectorSlot&);
                  // ^ overriding implicit one.
        void reset();
  private:
  ```

```
        InterruptHandler interruptHandler;
        bool enabled;
        bool autoReset;
        bool connected;
        InterruptVectorSlot(); // you cannot add hardware!
    };
```

4.3.1 BASIC SEMANTICS

The basic special semantics of each slot are the followings :

- When an interrupt occurs, the micro-controller senses some data bus lines that are interpreted as the offset of a desired slot (this offset is the interrupt identity).
- The slot content is interpreted as an interrupt handler, i.e. the address of an executable routine that expects one parameter (routine handled by Interrupt handlers are also called interrupt service routines, or ISRs).
- The micro-controller then suspends normal program execution, and calls the ISR, passing it the interrupt (i.e. the slot offset) as parameter.

Note : This call is performed in a context that is different from normal program execution.

- Whenever the ISR returns, the normal program execution resumes exactly where it was suspended (so, an ISR should always return).

An interrupt handler is the address of the code to be executed when the interrupt hit. At a C++ programming level, the interrupt handler is a pointer to a function that need a parameter of type `Interrupt` and return nothing.

- `typedef void (* InterruptHandler) (Interrupt);`

After writing your own ISR, assign each relevant slot as in following example :

```
void myISR(Interrupt);
ObaTargetMachine::interruptVector[IT_timer] = myISR;
```

Same interrupt service routine can be used for different interupts : the interrupt value can then be used inside the interrupt service routine block to differentiate interrupts treatment.

4.3.2 ENABLING/DISABLING INTERRUPTS

Interrupt Mask Registers are very special hardware registers, used as an array of bits, where each bit is a Boolean indicator for the corresponding interrupt (interrupt value equal to the index of the bit in the array).

The application programmer can enable desired interrupts after assign them an interrupt handler (as described above).

The interrupt enable mask register should be used to do that, as explained below :

- When a bit is set in the interrupt enable mask register, the matching interrupt is enabled,
- When the bit is cleared, this interrupt is disabled.

Initially, the address in each slot points to invalid address code. In order to prevent interrupts from hitting the program before safe interrupt handlers are set, all interrupts are initially disabled :any hit is then ignored.

Warning : ***Trying to enable an interrupt will stall the program if the interrupt is hit before assign an interrupt handler to the corresponding interrupt vector slot.***

There is also an auto-reset mask register:

- When a bit is set in the auto-reset mask, the corresponding handler is removed and its slot is turned back to stalling code address after the next hit of the corresponding interrupt.
- When that bit is cleared, the same interrupt handler remains in the slot after performance. Initially, for each interrupt the handler is permanent (cleared bit).

Interrupt mask values can be handled using bitwise operators (like '~' '&' '|' or '^').
There is some universal mask values, defined as constants.

```
typedef unsigned InterruptMask;

static const InterruptMask allInterrupts;

static const InterruptMask noInterrupt;
// ^ the initial value for both masks
```

Interrupt masks are to be applied upon the interrupt mask register of the target, using the assignment (“=”) operator.

```
class InterruptMaskRegister {
…
public:
   void operator= (InterruptMask itm);
…
}; // InterruptMaskRegister

typedef InterruptMaskRegister ItMReg;
// ^ short hand for convenience.
```

The 2 interrupt mask registers implemented in `OBATargetMachine` are :

- the enable interrupt mask register, which indicates which interrupts are enabled,
- the auto-reset interrupt mask register, which indicates which interrupts are auto-resettable.

```
static InterruptMaskRegister enabledInterruptsRegister;

static InterruptMaskRegister autoResetInterruptsRegister;
```

4.4 TIMER

The OBA target have a unique built-in timer : so, the virtual target gives only the same unique one.

This timer send an interrupt named `IT_timer` : after a `latencyInMillisecond` delay, the virtual target on which the calling program is running receives one `IT_timer` interrupt every `periodInMillisecond`.

Executing actions on timer interrupt needs to assign it an interrupt handler and to enable this interrupt.

Warning : ***There is actually no way to stop the timer after having started it.***

- ```
  static void setPeriodicTimer (
    int periodInMillisecond,
    int latencyInMillisecond);
  ```

4.5 REAL-TIME MONITOR FACILITIES

The OBA Target Machine (and the virtual target too) is to be considered as a quite bare micro-controller board. It does not give any multi-thread support from any real-time monitor (multi-threading is actually not a need for achieving a most efficient OBA design).

However, in the special case of multi-threading considered an educational goal, students that want to explore a multi-threaded design may ever (and can only) rely on such support taken directly from host OS API (Windows NT), or from language runtime library (Visual C++).

The alternate idea of writing their own real-time scheduler is unrealistic and deprecated, since micro-controller modeled in the virtual target has not been led up to emulation of registers, and doesn't allow for programming of context switching.

Nevertheless, the following primitive is introduced for convenience, as a useful tuning facility, since hardware threads related to interrupt handlers are implemented as host machine software threads for actual execution on the host machine. It suspends the calling thread for the time specified by the parameter, leaving free the CPU resource for other threads.

- ```
  static void idleWait (int delayInMillisecond);
  ```

4.6 OBA SPECIFIC BSP (BOARD SUPPORT PACKAGE).

The two enum types below provide symbolic names respectively identifying interrupts and related interrupt masks that are relevant to the user of the OBA Target Machine (i.e. which is bound to some specific devices).

The symbolic name of each indicate to OBA programmers which is the related environment device. These names and the values they represent came from interfaces design decisions, written in an Interface Design Document that is outside the scope of this case study.

- ```
  enum InterruptId    { IT_timer            =  5
                      , IT_drivingStation   = 14
                      , IT_controlPanel     = 15
                      };
  ```

- ```
  enum InterruptMaskId    {
                  atTimerInterrupt          = 0x0020 // 2**5
                , atDrivingStationInterrupt = 0x4000 // 2**14
  ```

```
              , atControlPanelInterrupt   = 0x8000 // 2**15
              , allObaRelatedInterrupts   =
                       atTimerInterrupt
                     | atDrivingStationInterrupt
                     | atControlPanelInterrupt
};
```

5. PRACTICAL USE

5.1 PRIVATE PARTS OF `ObaTargetMachine`

Reading anything in the private parts of this class is a user loss of time (as usual). Moreover, the private parts simulates mechanisms normally implemented by hardware, that are not soft changing.

5.2 SHORT-HANDS, FOR CONVENIENCE

There is a set of exported shorthand for convenience : these clauses and names provides direct visibility on BSP stuff, allowing the programmer not to worry about forgetting the boring qualifier and scope operator prefix.

- `typedef OBATargetMachine    OTM      ;`
- `using OTM::IT_timer                  ;`
- `using OTM::IT_drivingStation         ;`
- `using OTM::IT_controlPanel           ;`
- `using OTM::atTimerInterrupt          ;`
- `using OTM::atDrivingStationInterrupt ;`
- `using OTM::atControlPanelInterrupt   ;`
- `using OTM::allObaRelatedInterrupts   ;`

5.3 INCLUDE FILE

Header file `OBATargetMachine.h` is the file to be included in each application source file that need visibility on the target resources.

5.4 LIBRARIES

Each application that need access to a target (typically , OBA) shall link with the static library of the Virtual Target API : `%OBA_VTS%\OBAVirtualTarget.lib`

5.5 RESTRICTIONS OF USE

5.5.1 NO USE OF GLOBAL STATIC DATA

The C++ programming language does not provides means to determine the order of initialization of global static data. On its side however, the class `OBATargetMachine`, as mimicking a classical Operating System API, makes use of such global static data (enforcing uniqueness of corresponding system resources).

On the application side, OBA programmers also may wish to make use of global static data. They are allowed to do so, but as a consequence they must be very careful **NOT** to make their global static data depend on any `OBATargetMachine` feature for initialization. This is especially true for objects, that must not perform any `OBATargetMachine` call in their invoked constructor, nor in the constructor of any subobject (or base class) thereof, at any level of depth.

Warning : ***Failure to do so leads to unpredictable execution behaviour.***

The safer way is: avoid unnecessary global static objects. However there is a broad chance to need such a dangerous feature in a normal OBA application. Whenever an object absolutely needs global scope (in a file or externally among files), the classical solution is to declare a global static pointer instead of the object, then create the needed object through the pointer by dynamic allocation in the main or later on (that is, at a point where all global static initializations are certainly over), then use reference declarations to rename the resulting created objects wherever convenient.

6. NOTES

Not Used

APPENDIX A

Not Used